

ABSTRACT

The growing concerns over security vulnerabilities in conventional ATM systems, such as card theft and PIN shoulder-surfing, demand more reliable authentication methods. This study presents *Face ATM*, a two-factor banking interface that integrates facial biometrics with traditional PIN verification. The system is implemented using Flask as the backend framework and incorporates real-time face detection via Mediapipe, embedding extraction using a Torch-based OpenFace-style model, and classification with a Support Vector Machine (SVM). Webcam-based image capture enables user enrolment with multiple face samples, while persistent storage is managed through CSV files for account and transaction details. To validate identity, the system first performs face recognition, followed by PIN verification for defence-in-depth authentication. Experimental use demonstrates that the system reliably recognizes enrolled users under standard conditions and successfully supports basic banking operations such as deposits, withdrawals, and balance inquiries. Although limited by the absence of liveness detection and secure credential hashing, this work provides a reproducible prototype that demonstrates the practicality of lightweight, two-factor biometric ATM systems. The research underscores the potential of combining machine learning-based face recognition with conventional PINs to enhance banking security and usability.

Keywords: Face Recognition, Two-Factor Authentication, Mediapipe, Support Vector Machine (SVM), Flask, Biometric ATM, Deep Learning

CHAPTER-1

INTRODUCTION

1.1. Project Description

The Face ATM project is developed to make the process of using an ATM more secure and simple for users. Most people are familiar with how traditional ATMs work—you insert your card, punch in your PIN, and access your account. But that method isn't foolproof. This application is made using Flask and has a clean and simple web page that anyone can use.

This prototype uses a regular webcam to make sign-in feel more personal and secure: during registration you take at least five clear face photos.

Experimental use demonstrates that the system reliably recognizes enrolled users under standard conditions and successfully supports basic banking operations such as deposits, withdrawals, and balance inquiries. Although limited by the absence of liveness detection and secure credential hashing, this work provides a reproducible prototype that demonstrates the practicality of lightweight, two-factor biometric ATM systems. The research underscores the potential of combining machine learning-based face recognition with conventional PINs to enhance banking security and usability.

enter a few basic details and pick a 4-digit PIN; the photos are saved and a small recognition model is trained and stored for future logins, where the app first scans your face and—if it finds a match—prompts for the PIN, granting access only when both checks succeed; once signed in you can deposit, withdraw, view your balance and past transactions, with account data kept in CSV files, and the system intentionally remains a demonstration (no liveness detection and recognition quality varies with photo quality and camera/settings).

1.2 Scope of The Project

A simple ATM-style web app that uses a regular webcam to log users in with face recognition plus a PIN. Users register with photos and a 4-digit PIN; models are trained from those photos so the system can identify them later. After successful face + PIN verification, users can access a personal dashboard to deposit, withdraw, view balance, and see transaction history.

CHAPTER-2

LITERATURE SURVEY

2.1. Existing and Proposed System

Existing System

Over the last few years, researchers have made considerable progress in the field of deepfake and fake facial image detection. However, many existing approaches still face several limitations:

Reliance on a single model: A majority of detection systems depend on only one deep learning model. This reduces their ability to generalize across different datasets and lowers the overall accuracy.

Limited preprocessing techniques: In many systems, raw images are directly fed into the models without sufficient preprocessing. As a result, important image features, such as fine texture details, are not fully utilized.

Lack of accessibility: Several existing methods are designed for research purposes only and do not provide a user-friendly interface for practical use.

Key studies in the domain highlight these challenges:

Kim and Cho (2021): Proposed a deep learning system that extracts both content and manipulation-trace features using multi-channel convolution. They also introduced Class Activation Maps (CAM) for visualizing manipulated regions.

Jain, Ross, and Pankanti (2006): Explored biometric verification using both physical and behavioral characteristics, emphasizing challenges related to privacy and security.

Khodabakhsh et al. (2018): Demonstrated that traditional detection methods lose effectiveness when applied to new datasets, stressing the need for cross-dataset validation.

Thies et al. (2016): Designed a real-time facial reenactment framework with potential applications in VR, teleconferencing, and video dubbing, showing how manipulations can be created convincingly.

In summary, while existing systems have laid a strong foundation, their shortcomings in robustness, usability, and scalability leave room for significant improvement.

Proposed System

The proposed *Face ATM* system is designed to make ATM transactions more secure by combining facial recognition with PIN-based authentication for two-factor verification. Traditional ATMs rely only on cards and PINs, which can be lost, stolen, or observed. This system adds a biometric layer to reduce such risks.

Key features of the proposed system:

- Face-based authentication: Users register by capturing multiple face images through a webcam. Mediapipe detects and crops the face from each image.
- Feature extraction with Torch model: A pre-trained OpenFace-style model (nn4.small2.v1.t7) generates embeddings (numerical feature vectors) representing each user's face.
- Classification with SVM: The system uses a Support Vector Machine to identify users based on embeddings. Only recognized users can proceed.
- Two-factor login: After successful face recognition, users must enter their PIN to access their account.
- Banking operations: Users can perform deposits, withdrawals, check balances, and view transaction history.
- Persistent storage: Accounts and transactions are stored in CSV files, and trained models are saved using pickles for reuse.
- Web interface: A Flask-based interface provides an intuitive and easy-to-use platform for registration, login, and banking functions.

This system demonstrates how combining biometrics with traditional PIN authentication can improve security while maintaining simplicity and usability.

2.2. Feasibility Study

Technical Feasibility

The project can be implemented using widely available software tools and standard hardware:

Web development: Flask is used to handle backend logic and serve the user interface.

Face detection: Mediapipe detects faces in real-time using a standard webcam.

Feature extraction and classification: The Torch model generates embeddings, and SVM classifies them to identify users.

Hardware compatibility: The system works on normal computers without needing specialized devices, and real-time performance is achievable with CPUs.

Economic Feasibility

The project is cost-effective because of Open-source tools: All required libraries (Flask, Mediapipe, Torch, scikit-learn, OpenCV, Pandas, Numpy) are free. Standard hardware: No expensive GPUs or specialized equipment are necessary. Scalability: Multiple users can be supported without significant extra costs, though CSV-based storage has limits for large-scale deployments.

2.3. Tools and Technologies Used

Category	Technology	Purpose
Backend Framework	Flask	Web application development
Face Detection	Mediapipe	Detecting and cropping faces
Feature Extraction	Torch (OpenFace)	Generating face embeddings
Classifier	SVM (scikit-learn)	Identifying users from embeddings
Image Processing	OpenCV	Handling image capture and preprocessing
Data Processing	Pandas, Numpy	Managing accounts and transactions
Frontend	HTML, CSS, JavaScript	User interface development
Model Storage	Pickle	Saving trained embeddings and classifier

Table 2.1: Tools and Technologies Used

2.4. Hardware and Software Requirements

Hardware Requirements

Component	Minimum Specification	Recommended Specification
Processor	Intel Core i3 or equivalent	Intel Core i5 or higher
RAM	4 GB	8 GB or higher
Storage	2 GB free space	10 GB or higher
Webcam	Built-in or USB webcam	HD webcam for better image quality

Table 2.2: Hardware Requirements

Software Requirements

Software	Version	Purpose
Python	3.7 or higher	Programming language
Flask	2.x	Web framework
Mediapipe	Latest version	Real-time face detection
Torch	1.x	Generating face embeddings
scikit-learn	Latest version	SVM classifier
OpenCV	4.x	Image capture and preprocessing
Pandas/Numpy	Latest versions	Managing CSV files and computations
Web Browser	Latest versions	Accessing the web interface

Table 2.3: Software Requirements

CHAPTER-3

SOFTWARE REQUIREMENTS SPECIFICATIONS

3.1 Users

The Face ATM system is designed for different categories of users, each with specific needs and usage patterns:

The system is designed to serve different types of users, each with their own needs and usage patterns. Bank customers use it most frequently, performing ATM transactions on a daily basis. For them, secure login, quick access, and a simple interface are essential to ensure smooth transactions. Bank staff interact with the system on a weekly basis, mainly to manage accounts and monitor transaction logs through a clear and functional dashboard. System administrators use the platform less often, usually on a monthly basis, to handle backend tasks such as user monitoring, security updates, and error handling. Finally, researchers and developers access the system occasionally, focusing on analyzing model outputs, studying embeddings, and reviewing logs to enhance biometric authentication techniques. This division ensures that the system effectively meets the unique requirements of every user group.

3.2 Functional Requirements

The system includes several functional modules necessary for secure and smooth operation:

FR1: User Management System

Requirement ID	Description	Priority	Status
FR1.1	User registration with account details	High	Implemented
FR1.2	Secure login using face + PIN	High	Implemented
FR1.3	Session management with timeout	High	Implemented
FR1.4	User profile updates and management	Medium	Implemented

Table 3.1: FR1 User Management System

3.3 Nonfunctional Requirements

- Runs locally (no special hardware).
- Lightweight, minimal dependencies.
- Reasonable performance for a small user set (single-machine use).
- Basic input validation and user feedback via the web UI.

CHAPTER-4

SYSTEM DESIGN

4.1 System Architecture

Model Validation: The SVM classifier is tested using previously captured embeddings to ensure accurate user identification. **Session Management:** User sessions are maintained during active operations and automatically timed out after inactivity to prevent unauthorized access. **Security Measures:** Sensitive operations, such as PIN entry and transaction updates, are validated server-side to prevent tampering.

4.1.2 Architecture Benefits

The Face ATM System's architecture has been carefully planned to provide a number of useful benefits that make it strong and flexible. Each of its essential components—face detection, feature extraction, classification, and banking operations—can operate independently thanks to its modular design. Because of this division, maintenance is made simpler, and updates and improvements can be implemented more easily without causing system-wide disruptions.

Scalability is yet another important advantage. The system's ability to easily accept new users without necessitating significant structural adjustments is crucial for practical implementation in expanding banking settings. A probability-based threshold in the face recognition process ensures robustness, reducing misclassification and increasing overall accuracy. Real-time face detection and embedding extraction are built into the system to increase efficiency and facilitate rapid and seamless user authentication.

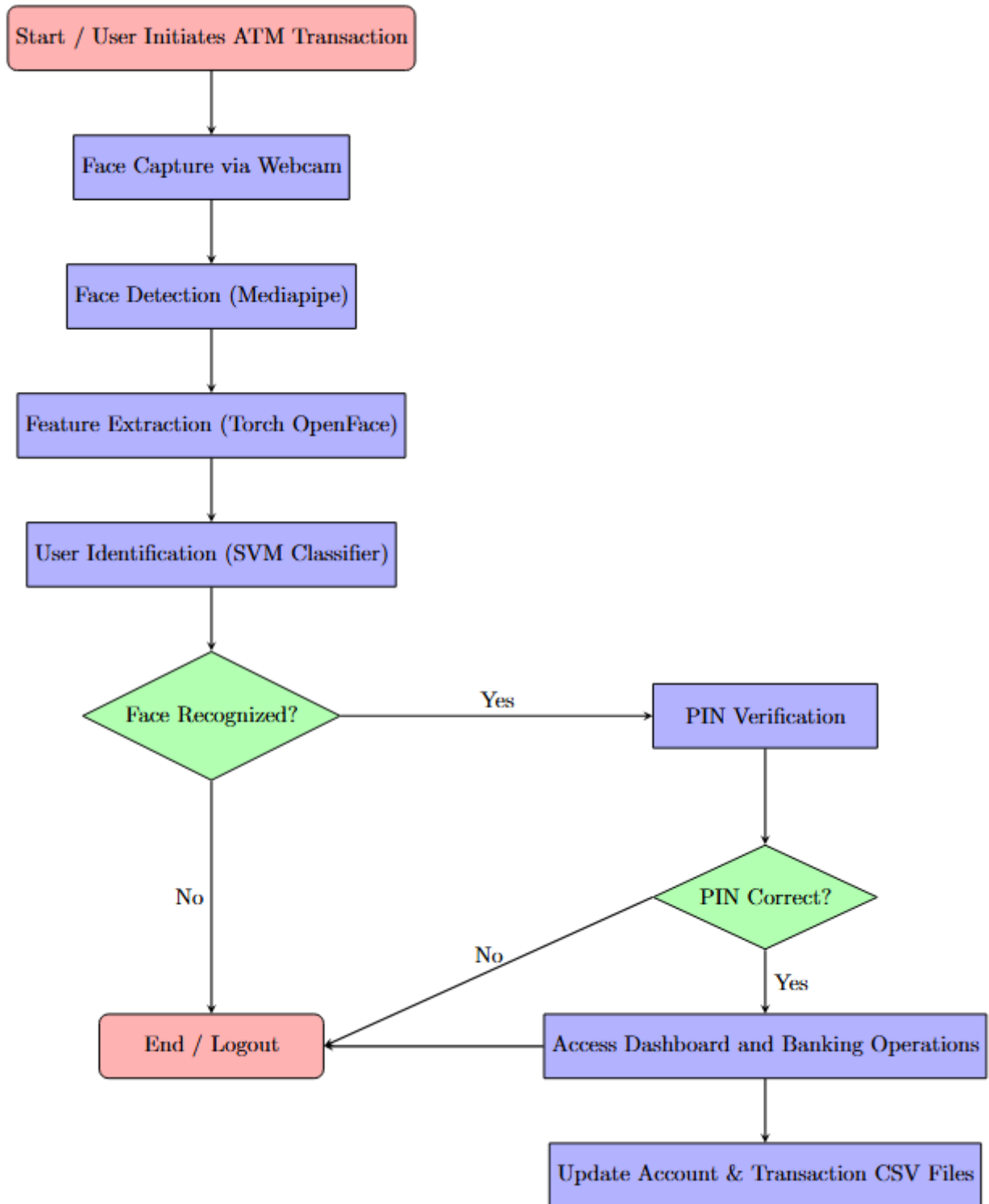


Figure 4.1: System Architecture Diagram

4.2 System Perspective

The *Face ATM* system functions as a standalone web-based banking application, integrating multiple components to provide secure and reliable ATM services. Its core The Face ATM system is a robust, standalone web-based banking application that seamlessly integrates biometric authentication with traditional banking operations.

This architecture provides a workable and scalable solution for biometric-based banking services by guaranteeing that the Face ATM system stays safe, responsive, and dependable.

4.3 Context Diagram

The **context diagram** provides a high-level view of how the Face ATM system interacts with external entities, data sources, and storage components:

- **External Entities:**
 - **Users/Customers:** Register, authenticate, and perform banking operations.
 - **Bank Administrators:** Monitor accounts, manage users, and oversee system performance.
- **Core System Functions:**
 - **User Registration and Authentication:** Captures facial images, stores embeddings, verifies identity with SVM + PIN.
 - **Banking Operations:** Deposit, withdrawal, balance check, and transaction history.
 - **Session Management:** Ensures secure, timed sessions.
- **Supporting Storage Systems:**
 - **Account and Transaction CSV files:** Stores account details and transaction history.
 - **Model Repository:** Saves trained embeddings and SVM classifiers for recognition.
 - **Temporary Image Storage:** Holds captured images during registration or verification.
- **Data Flow:**
 - Users interact with the web interface to provide input (face + PIN).
 - The system processes the input through the face recognition and authentication pipeline.
 - Banking operations update account and transaction records in CSV files.
 - Outputs are displayed to the user as dashboard information and operation confirmations.

Overall, the context diagram demonstrates how the Face ATM system securely connects users, data storage, and the processing pipeline to provide efficient, reliable, and accurate authentication and banking services.

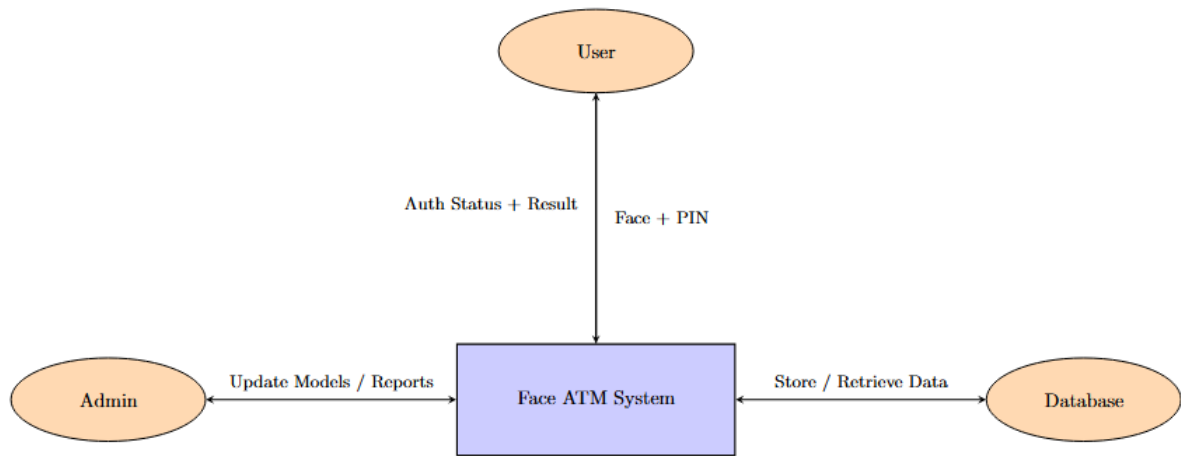


Figure 4.2: Context Diagram

CHAPTER-5

DETAILED DESIGN

This section presents the **detailed design of the Face ATM system**, including data flow diagrams, activity diagrams, use case diagrams, and sequence diagrams. These models describe the functional behavior, process interactions, and data flow within the system.

5.1 Data Flow Diagram (DFD)

The **Data Flow Diagram** models the logical flow of data within the Face ATM system. It provides both a high-level overview (Level 0) and a detailed process decomposition (Level 1), illustrating how data moves through different components.

5.1.1 Level 0 DFD

The **Level 0 DFD** represents the system as a single process interacting with external entities.

1. External Entities:

- **User/Customer:** Primary stakeholder who registers, authenticates, and performs banking operations.
- **Bank Administrator:** Manages accounts, monitors system performance, and updates configurations.

2. Central Procedure:

- **Face ATM System:** A single process encapsulating the entire application.

3. . Data Flows:

The system controls an organized data flow between various components and users. Users interact with the system by entering their login or registration credentials and providing a webcam-captured image of their face for biometric authentication. They can send banking requests, like deposits, withdrawals, or balance inquiries, after logging in. The user receives feedback from the system in the form of banking responses, such as updated balances or transaction confirmations, as well as authentication status, which indicates whether the login was successful or not. Administrators send configuration inputs, such as model updates and

settings modifications, to the system.

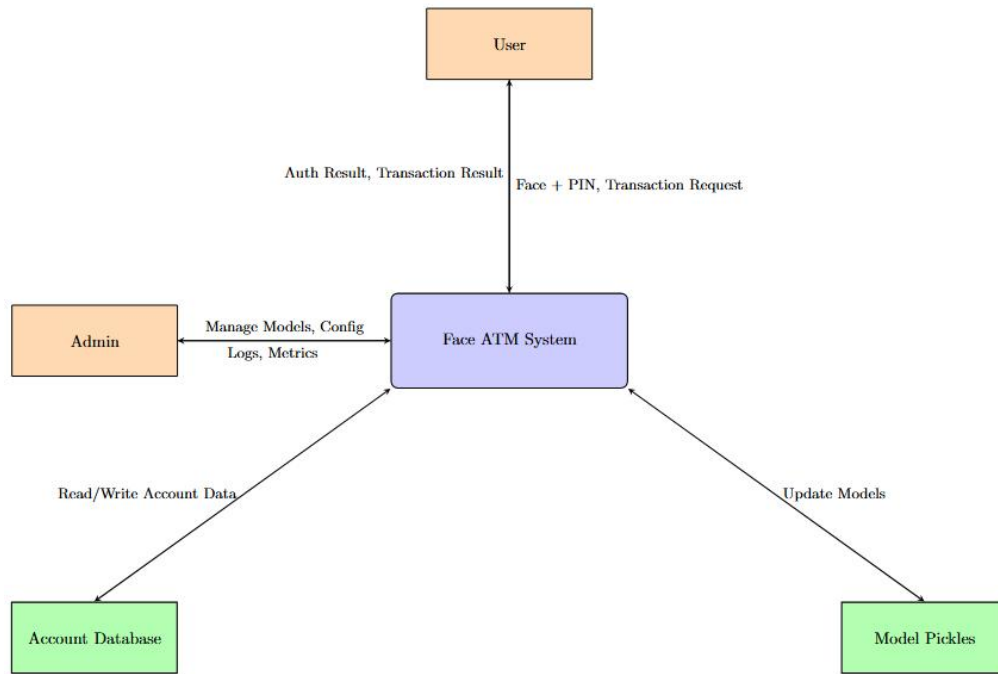


Figure 5.1: Data Flow Diagram Level 0

5.2 Activity Diagram

From user registration to secure transaction processing, the Face ATM system operates according to a streamlined workflow. A user first registers by supplying a personal identification number (PIN) and a picture of their face. In order to identify the new user, the system modifies its classifier model and safely stores this data. The user enters their PIN and displays their face when logging in. To finish the authentication process, the system uses facial recognition to confirm the user's identity first, then PIN validation. The user can perform banking functions like making deposits, taking out cash, and checking their account balance after being authenticated. To ensure transparency and real-time feedback, the system updates the pertinent records following each transaction and shows the result on the user dashboard.

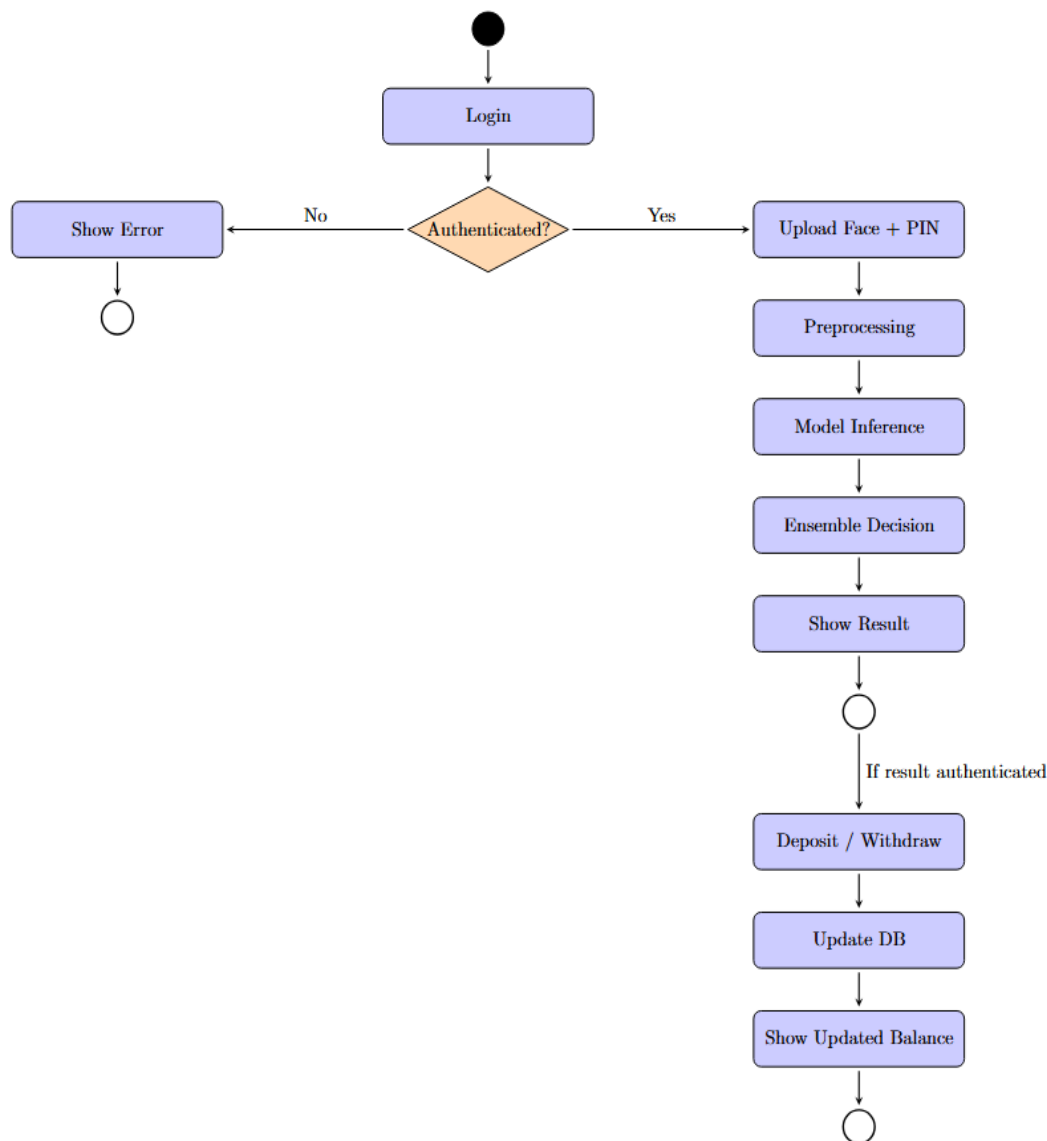


Figure 5.2: Activity Diagram

5.3 Use Case Diagram

The **use case diagram** illustrates the functionality through two primary actors:

- **Admin:** Manages accounts, monitors system health, updates models, and performs maintenance.
- **User:** Registers, authenticates, and performs banking operations via the dashboard.

5.2.1 Use case diagram Admin

Admin: Manages accounts, monitors system health, updates models, and performs maintenance.

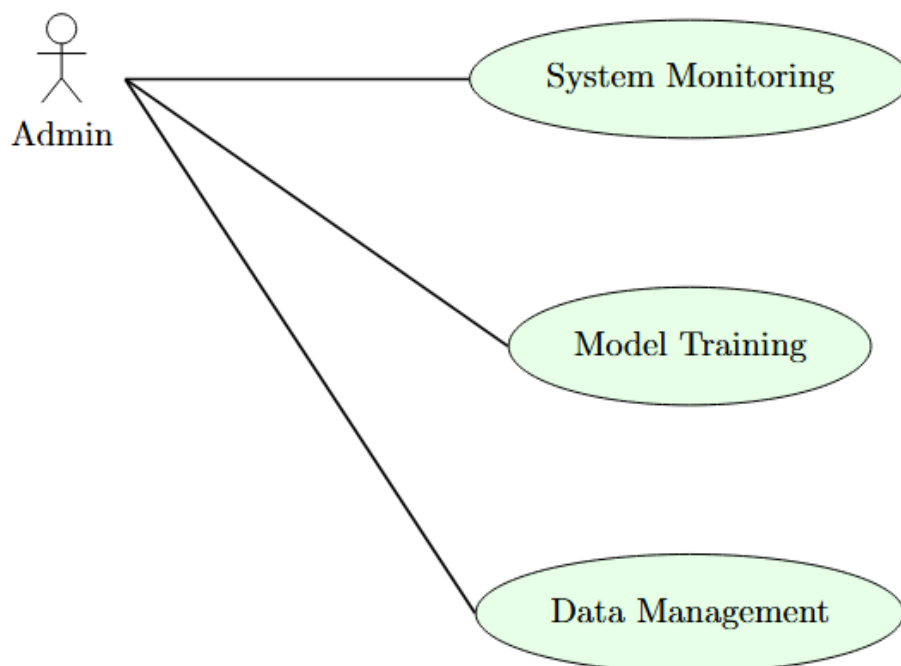


Figure 5.3: Use-case diagram Admin

5.2.2 Use case Diagram User

User: Registers, authenticates, and performs banking operations via the dashboard.

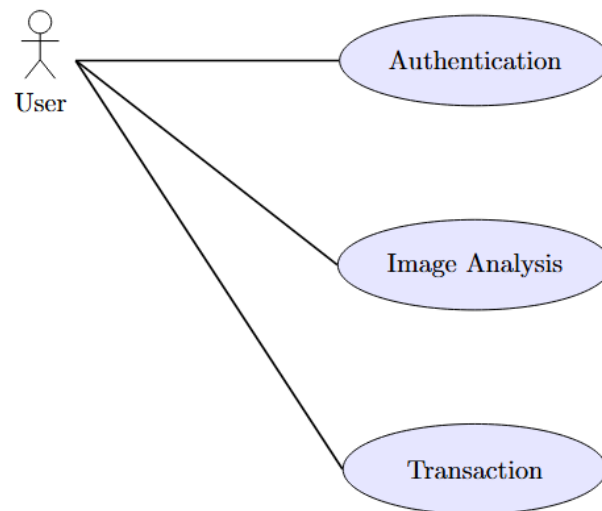


Figure 5.4: Use-case diagram User

5.3.1 Sequence Diagram

The **sequence diagram** illustrates the interaction between system components in **two main phases**:

1. Authentication Phase

- User submits face + PIN.
- System captures image, preprocesses it, and generates embeddings.
- SVM classifier predicts identity.
- System verifies PIN and grants access.

2. Banking Operation Phase

- Authenticated user requests a banking operation.
- System updates CSV files with transaction details.
- Dashboard displays updated balance and confirmation.

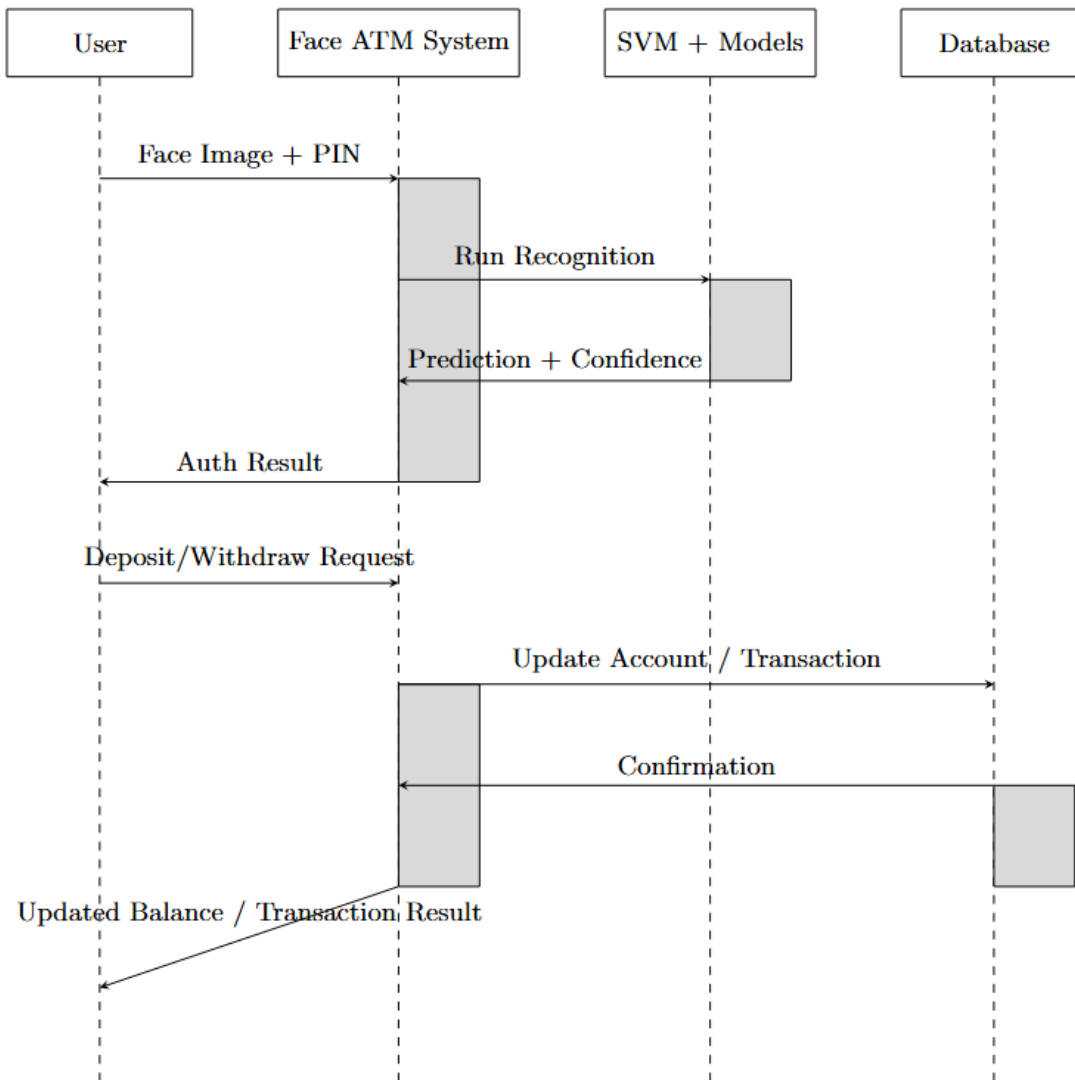


Figure 5.5: Sequence Diagram

CHAPTER-6

IMPLEMENTATION

6.1 Coding Standards

The implementation of the Face ATM DeepFake Detection System follows established Python programming conventions to ensure the code is readable, maintainable, and scalable. The key coding standards adopted are summarized below

The project follows a thoughtful and well-structured approach to coding standards that makes collaboration and maintenance much smoother. It sticks to PEP 8 guidelines, using a clean 4-space indentation and keeping line lengths under control for better readability. Naming conventions are intuitive, with variables and functions written in snake-case—so names like `preprocess_image()` or `model_predictions()` instantly tell you what they're about. Every function includes a clear docstring that explains what it does, what inputs it expects, and what it returns, making the code easy to understand even for someone new to the project. Error handling is taken seriously, with try-except blocks used throughout to catch issues gracefully and log them for review. The code is also neatly organized into separate modules, with different Python files handling preprocessing, model training, inference, and the user interface. This modular setup keeps things tidy and makes it easier to scale or update individual components without disrupting the whole system.

6.2 Code Implementation

6.2.1 app.py Code

```
app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 512 * 1024 * 1024 # 512 MB
import secrets
app.secret_key = os.environ.get("FLASK_SECRET_KEY", secrets.token_hex(16))

# initial load
embedder, recognizer, le = load_models()

@app.route("/")
```

```
def home():
    return render_template("index.html")

# Login: Step 1 - Face Verification (via /verify and /api/verify)
@app.route("/login", methods=["GET"])
def login():
    return render_template("verify.html")

@app.route("/api/verify", methods=["POST"])
def api_verify():
    global embedder, recognizer, le
    try:
        data = request.json
        img_b64 = data.get("image")
        result = recognize_from_image_b64(img_b64, embedder, recognizer, le)
        if result.get("name") != "unknown":
            session["user_id"] = result["name"]
            return jsonify(result)
    except Exception as e:
        return jsonify({"ok": False, "error": "No face matched or server error."}), 200

# Login: Step 2 - PIN/Password Verification
@app.route("/login_pin", methods=["GET", "POST"])
def login_pin():
    user_id = session.get("user_id")
    print('login_pin route called, user_id:', user_id)
    if not user_id:
        return redirect(url_for("login"))
    if request.method == "GET":
        return render_template("login_pin.html", user_id=user_id)
    pin = request.form.get("pin")
    print('PIN received:', repr(pin))
```

```

df = read_accounts()
row = df[df["unique_id"].astype(str) == str(user_id)]
expected_pin = str(row.iloc[0]["password"]) if not row.empty else None
print('Expected PIN:', repr(expected_pin))
if row.empty or expected_pin != pin:
    flash("Invalid PIN/password", "danger")
    print('Invalid PIN/password for user_id:', user_id)
    return render_template("login_pin.html", user_id=user_id)
session["logged_in"] = True
return redirect(url_for("dashboard", user_id=user_id))

# ...existing code...

```

```

@app.route("/dashboard/<user_id>")
def dashboard(user_id):
    if not session.get("logged_in") or session.get("user_id") != user_id:
        flash("Please login first.", "danger")
        return redirect(url_for("home"))
    df = read_accounts()
    row = df[df["unique_id"].astype(str) == str(user_id)]
    if row.empty:
        flash("Account not found", "danger")
        return redirect(url_for("home"))
    acc_no = row.iloc[0]["account_number"]
    name = row.iloc[0]["name"]
    bal = int(row.iloc[0]["account_balance"])
    history = get_history(user_id)
    return render_template("dashboard.html", user_id=user_id, acc_no=acc_no, name=name,
bal=bal, history=history)

```

```

@app.route("/deposit/<user_id>", methods=["POST"])
def deposit(user_id):
    if not session.get("logged_in") or session.get("user_id") != user_id:
        flash("Please login first.", "danger")

```

```

    return redirect(url_for("home"))
amt = int(request.form.get("amount",0))
if amt <= 0:
    flash("Enter valid amount", "danger")
    return redirect(url_for("dashboard", user_id=user_id))
df = read_accounts()
df["account_balance"] = df["account_balance"].astype(int)
df.loc[df["unique_id"].astype(str) == str(user_id), "account_balance"] += amt
write_accounts(df)
log_transaction(user_id, "Deposit", amt)
flash(f"₹{amt} deposited", "success")
return redirect(url_for("dashboard", user_id=user_id))

```

```

@app.route("/withdraw/<user_id>", methods=["POST"])
def withdraw(user_id):
    if not session.get("logged_in") or session.get("user_id") != user_id:
        flash("Please login first.", "danger")
        return redirect(url_for("home"))
    amt = int(request.form.get("amount",0))
    if amt <= 0:
        flash("Enter valid amount", "danger")
        return redirect(url_for("dashboard", user_id=user_id))
    df = read_accounts()
    df["account_balance"] = df["account_balance"].astype(int)
    bal_row = df.loc[df["unique_id"].astype(str) == str(user_id), "account_balance"]
    if bal_row.empty:
        flash("Account not found", "danger")
        return redirect(url_for("dashboard", user_id=user_id))
    bal = int(bal_row.values[0])
    if amt > bal:
        flash("Insufficient balance", "danger")
        return redirect(url_for("dashboard", user_id=user_id))
    df.loc[df["unique_id"].astype(str) == str(user_id), "account_balance"] = bal - amt
    write_accounts(df)

```

```

    log_transaction(user_id, "Withdraw", amt)
    flash(f"₹{amt} withdrawn", "success")
    return redirect(url_for("dashboard", user_id=user_id))
# Logout
@app.route("/logout")
def logout():
    session.clear()
    flash("Logged out.", "info")
    return redirect(url_for("home"))

@app.route("/train", methods=["POST"])
def train_route():
    try:
        res = train_model()
        global embedder, recognizer, le
        embedder, recognizer, le = load_models()
        return jsonify({"ok": True, "trained": res.get("trained", 0)})
    except Exception as e:
        return jsonify({"ok": False, "error": str(e)}), 500

# Registration page
@app.route("/register", methods=["GET", "POST"])
def register():
    print('Register route called')
    if request.method == "GET":
        return render_template("register.html")
    print('Form data received')
    name = request.form.get("name")
    acc_no = request.form.get("acc_no") or str(secrets.randbelow(1000000000))
    password = request.form.get("password")
    images_json = request.form.get("images")
    # validate and parse deposit safely
    deposit_raw = request.form.get("deposit", "0")

```

```

try:
    deposit = int(deposit_raw)
except Exception:
    deposit = 0
images = []
try:
    if images_json:
        import base64, uuid, json
        images = json.loads(images_json)
        if not isinstance(images, list):
            images = []
    print('Images loaded:', len(images))
    if not (name and acc_no and deposit > 0 and password and images and len(images) >= 5):
        print('Validation failed')
        flash("All fields required and at least 5 face images", "danger")
        return redirect(url_for("register"))
    user_dir = os.path.join("dataset", acc_no)
    os.makedirs(user_dir, exist_ok=True)
    for idx, b64img in enumerate(images):
        if b64img.startswith("data:"):
            b64img = b64img.split(",")[1]
            imgdata = base64.b64decode(b64img)
            fname = f"face_{idx+1}_{uuid.uuid4().hex[:8]}.jpg"
            with open(os.path.join(user_dir, fname), "wb") as f:
                f.write(imgdata)
    print('Images saved')
    # Save account before training
    import pandas as pd
    df = read_accounts()
    new_row = {"unique_id": acc_no, "account_number": acc_no, "name": name, "bank":
"FaceATM", "password": password, "account_balance": deposit}
    df = pd.concat([df, pd.DataFrame([new_row])], ignore_index=True)
    print('Saving account:', new_row)
    write_accounts(df)

```

```
print('Account written to CSV')
# Train model for this user
try:
    train_model()
except Exception as e:
    print("Training failed:", e)
    flash(f'Account saved, but training failed: {e}', "warning")
    return redirect(url_for("home"))
flash("Account created! You can now login.", "success")
return redirect(url_for("home"))
except Exception as e:
    print('Registration failed:', e)
    flash(f'Registration failed: {e}', "danger")
    return redirect(url_for("register"))

@app.route("/enroll", methods=["GET", "POST"])
def enroll():
    if request.method == "GET":
        return render_template("enroll.html")
    user_id = request.form.get("user_id")
    if not user_id:
        flash("Enter user id", "danger")
        return redirect(url_for("enroll"))
    files = request.files.getlist("images")
    if not files:
        flash("Upload images", "danger")
        return redirect(url_for("enroll"))
    user_dir = os.path.join("dataset", str(user_id))
    os.makedirs(user_dir, exist_ok=True)
    count = 0
    for f in files:
        save_path = os.path.join(user_dir, f.filename)
        f.save(save_path)
        count += 1
```

```
flash(f'Saved {count} images for user {user_id}. Now click Train.', "success")
return redirect(url_for("enroll"))
```

```
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=5000)
```

6.2.2 MODEL TRAINING

```
# train.py
import os
import cv2
import pickle
import mediapipe as mp
from imutils import paths
from sklearn.preprocessing import LabelEncoder
from sklearn.svm import SVC

DATASET_DIR = "dataset"
OUTPUT_DIR = "output"
EMBEDDER_MODEL = "nn4.small2.v1.t7"

def train_model():
    if not os.path.exists(EMBEDDER_MODEL):
        raise FileNotFoundError(f'Missing embedder file: {EMBEDDER_MODEL}')
    embedder = cv2.dnn.readNetFromTorch(EMBEDDER_MODEL)
    imagePaths = list(paths.list_images(DATASET_DIR))
    if not imagePaths:
        raise FileNotFoundError("No images in dataset to train.")
    mp_fd = mp.solutions.face_detection.FaceDetection(min_detection_confidence=0.5)
    knownEmbeddings = []
    knownNames = []
    total = 0
    for imagePath in imagePaths:
        name = imagePath.split(os.path.sep)[-2]
```

```

image = cv2.imread(imagePath)
if image is None:
    continue
rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
results = mp_fd.process(rgb)
if results.detections:
    for det in results.detections:
        bbox = det.location_data.relative_bounding_box
        ih, iw, _ = image.shape
        x = max(0, int(bbox.xmin * iw))
        y = max(0, int(bbox.ymin * ih))
        w = int(bbox.width * iw)
        h = int(bbox.height * ih)
        x2 = min(iw, x + w)
        y2 = min(ih, y + h)
        face = image[y:y2, x:x2]
        if face.size == 0:
            continue
        (fH, fW) = face.shape[:2]
        if fW < 20 or fH < 20:
            continue
        faceBlob = cv2.dnn.blobFromImage(face, 1.0/255, (96,96), (0,0,0), swapRB=True,
crop=False)
        embedder.setInput(faceBlob)
        vec = embedder.forward()
        knownNames.append(name)
        knownEmbeddings.append(vec.flatten())
        total += 1
if total == 0:
    raise ValueError("No valid faces found.")
os.makedirs(OUTPUT_DIR, exist_ok=True)
data = {"embeddings": knownEmbeddings, "names": knownNames}
with open(os.path.join(OUTPUT_DIR, "embeddings.pickle"), "wb") as f:
    f.write(pickle.dumps(data))

```

```

le = LabelEncoder()
labels = le.fit_transform(knownNames)
recognizer = SVC(C=1.0, kernel="linear", probability=True)
recognizer.fit(knownEmbeddings, labels)
with open(os.path.join(OUTPUT_DIR, "recognizer.pickle"), "wb") as f:
    f.write(pickle.dumps(recognizer))
with open(os.path.join(OUTPUT_DIR, "le.pickle"), "wb") as f:
    f.write(pickle.dumps(le))
return {"trained": total}

if __name__ == "__main__":
    print("Training... this may take a while")
    res = train_model()
    print("Trained:", res)

```

6.2.3 MODEL TESTING

```

# face_utils.py
import os
import cv2
import numpy as np
import pickle
import mediapipe as mp

# Paths - adjust if filenames differ
OUTPUT_DIR = "output"
EMBEDDER_MODEL = "nn4.small2.v1.t7" # if your embedder is named differently,
change
CAFFE_PROTO = "deploy.prototxt"
CAFFE_MODEL = "res10_300x300_ssd_iter_140000.caffemodel"

mp_fd = mp.solutions.face_detection.FaceDetection(min_detection_confidence=0.5)

def load_models():
    embedder = None
    recognizer = None
    le = None
    if os.path.exists(EMBEDDER_MODEL):
        embedder = cv2.dnn.readNetFromTorch(EMBEDDER_MODEL)
    rec_path = os.path.join(OUTPUT_DIR, "recognizer.pickle")
    le_path = os.path.join(OUTPUT_DIR, "le.pickle")
    if os.path.exists(rec_path) and os.path.exists(le_path):
        recognizer = pickle.loads(open(rec_path, "rb").read())

```

```

    le = pickle.loads(open(le_path, "rb").read())
    return embedder, recognizer, le

def detect_faces_mediapipe(frame):
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = mp_fd.process(rgb)
    boxes = []
    (h,w) = frame.shape[:2]
    if results.detections:
        for det in results.detections:
            bbox = det.location_data.relative_bounding_box
            x = max(0, int(bbox.xmin * w))
            y = max(0, int(bbox.ymin * h))
            fw = int(bbox.width * w)
            fh = int(bbox.height * h)
            x2 = min(w, x + fw)
            y2 = min(h, y + fh)
            boxes.append((x,y,x2,y2))
    return boxes

def extract_embedding(embedder, face):
    if face is None:
        print('extract_embedding: face is None')
        return None
    if not isinstance(face, np.ndarray):
        print('extract_embedding: face is not a numpy array, type:', type(face))
        return None
    print('extract_embedding: face shape:', face.shape, 'dtype:', face.dtype)
    blob = cv2.dnn.blobFromImage(face, 1.0/255, (96,96), (0,0,0), swapRB=True, crop=False)
    embedder.setInput(blob)
    vec = embedder.forward()
    return vec.flatten()

def recognize_from_frame(frame, embedder, recognizer, le, min_proba=0.5):
    """
    Input: BGR frame (numpy)
    Output: dict {name, proba} or {'name':'unknown'}
    """
    boxes = detect_faces_mediapipe(frame)
    if not boxes:
        return {"name":"unknown","proba":0.0}
    # Take largest face
    boxes = sorted(boxes, key=lambda b: (b[2]-b[0])*(b[3]-b[1]), reverse=True)
    startX,startY,endX,endY = boxes[0]
    face = frame[startY:endY, startX:endX]
    (fH,fW) = face.shape[:2]
    if fH < 20 or fW < 20:
        return {"name":"unknown","proba":0.0}
    if embedder is None or recognizer is None or le is None:
        return {"name":"unknown","proba":0.0}

```

```

vec = extract_embedding(embedder, face)
preds = recognizer.predict_proba([vec])[0]
print('Face verification: prediction probabilities:', preds)
print('Face verification: label mapping:', le.classes_)
j = np.argmax(preds)
proba = float(preds[j])
name = le.classes_[j]
if name == 'unknown' or proba < min_proba:
    return {"name": "unknown", "proba": proba}
return {"name": str(name), "proba": proba}

def recognize_from_image_b64(b64data, embedder, recognizer, le, min_proba=0.5):
    import base64
    if not b64data:
        return {"error": "no image"}
    if b64data.startswith("data:"):
        b64data = b64data.split(",")[1]
    try:
        imgdata = base64.b64decode(b64data)
    except Exception as e:
        return {"error": f"invalid base64: {e}"}
    nparr = np.frombuffer(imgdata, np.uint8)
    frame = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
    if frame is None:
        return {"error": "invalid image"}
    return recognize_from_frame(frame, embedder, recognizer, le, min_proba=min_proba)

```

6.3 SCREENSHOTS

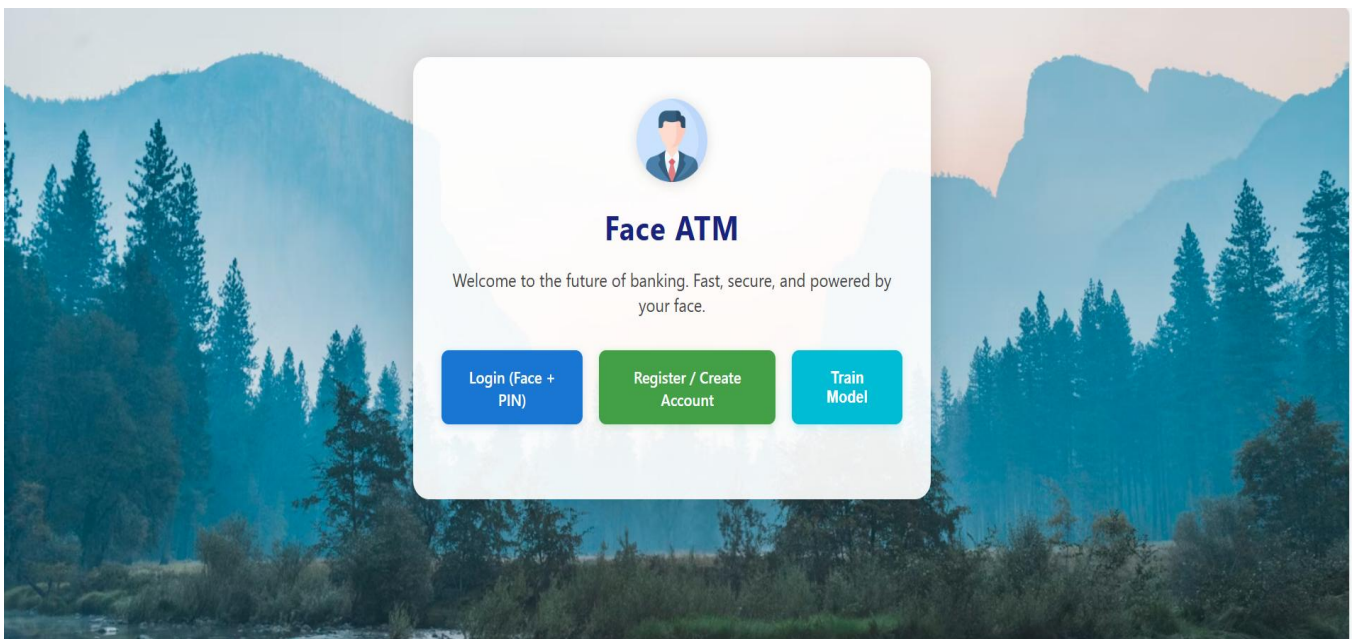
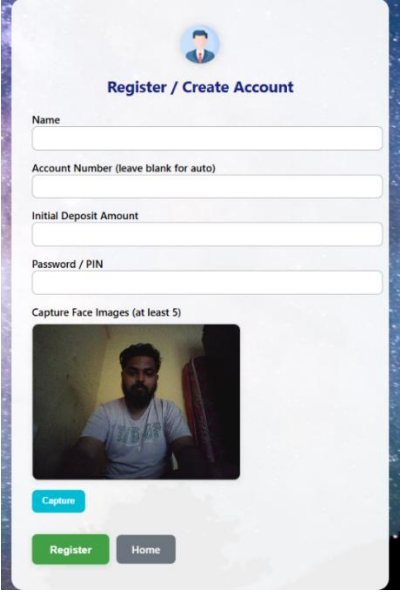
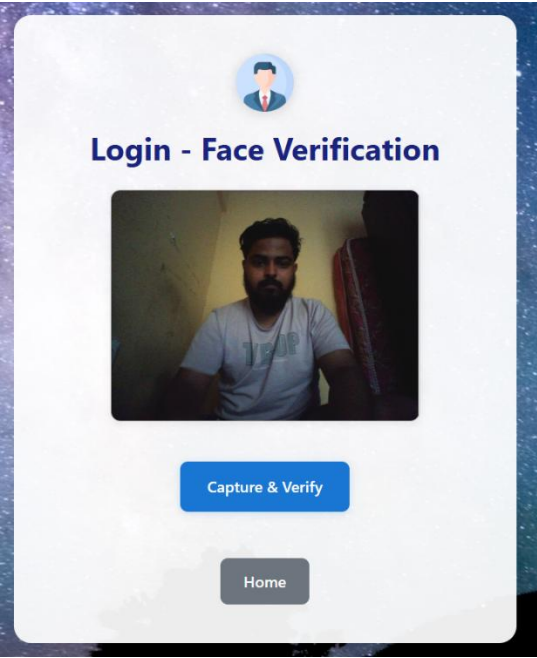


Figure 6.1: Home page



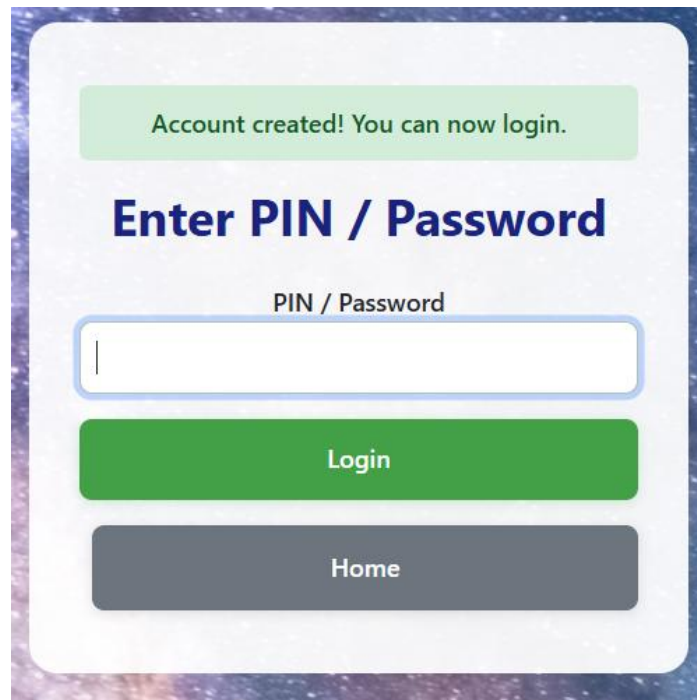
The registration page features a white background with a blue header bar. At the top center is a circular profile icon of a man with a beard. Below the icon is the title "Register / Create Account" in bold blue text. The form consists of several input fields: "Name", "Account Number (leave blank for auto)", "Initial Deposit Amount", and "Password / PIN". Below these fields is a section titled "Capture Face Images (at least 5)" which contains a video feed of a man's face. A small blue "Capture" button is positioned below the video feed. At the bottom of the form are two buttons: a green "Register" button and a grey "Home" button.

Figure 6.2: Registration page



The login-facial verification page has a white background with a blue header bar. At the top center is a circular profile icon of a man with a beard. Below the icon is the title "Login - Face Verification" in bold blue text. The main part of the page features a large video feed of a man's face. Below the video feed is a blue button labeled "Capture & Verify". At the bottom of the page is a grey button labeled "Home".

Figure 6.3: Login-Face verification



Account created! You can now login.

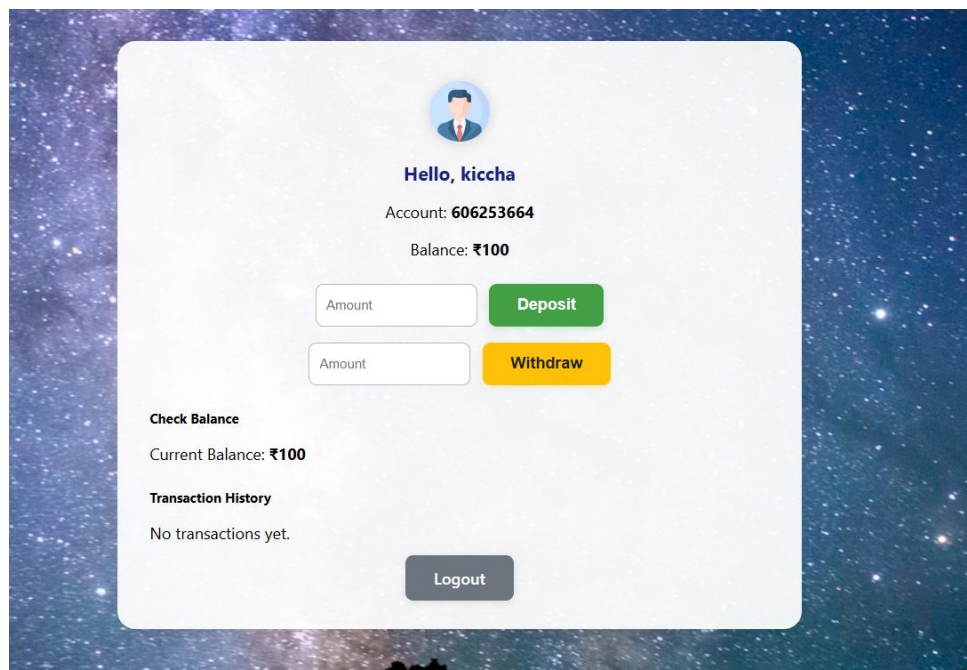
Enter PIN / Password


PIN / Password

Login

Home

Figure 6.4: login-pin or password




Hello, kiccha
Account: **606253664**
Balance: **₹100**

Amount **Deposit**

Amount **Withdraw**

Check Balance
Current Balance: **₹100**

Transaction History
No transactions yet.

Logout

Figure 6.5: Bank transaction page

CHAPTER 7

SOFTWARE TESTING

Software testing ensures that the Face ATM system works as expected, correctly authenticates users, and produces reliable results. Testing covers both frontend functionality (user login, registration, dashboard, PIN + face input) and backend deep learning model predictions. The main goals are error identification, correctness verification, and system reliability, security, and usability.

7.1 Unit Testing

Individual components were tested separately to ensure proper functionality:

Test Case	Input	Expected Output	Result
Valid Face Image Processing	JPG/PNG face image (256×256)	128×128×3 normalized array	Pass
Invalid Image Format	Text file (.txt)	None or error message	Pass
Gabor Filter Application	Grayscale face image	Filtered RGB image	Pass
Image Normalization	Raw pixel values (0–255)	Normalized values (0–1)	Pass
PIN Verification Function	Correct/Incorrect PIN	Authentication success/failure	Pass
Model Loading Tests	N/A	Models load within reasonable memory	Pass

Table 7.1: Unit Testing

7.2 Automation Testing

Testing the interaction between different system components:

Test Scenario	Components Tested	Expected Result	Status
User Login Flow	Face + PIN Authentication + Session Management	Successful login and redirect to dashboard	Pass
Face Image Upload & Processing	File Upload + Preprocessing + Model Inference	Complete prediction pipeline execution	Pass
Individual Model Prediction	Preprocessing + Selected Model	Correct user authentication result displayed	Pass
Error Handling	Invalid inputs + Exception handling	Graceful error messages displayed	Pass
Multi-User Access	Concurrent login attempts	All users processed without error	Pass

Table 7.2: Automation Testing

7.3 Test Cases

Test cases were designed to check the critical functionalities of the system. Some representative test cases are given below:

Test ID	Test Description	Steps	Expected Result	Status
TC001	User Registration	1. Navigate to registration page → 2. Enter valid details (Name, Email, PIN, Face Image) → 3. Submit f	Account created successfully, redirect to login	Pass
TC002	User Login	1. Enter registered email → 2. Upload registered face image → 3. Enter PIN → 4. Click Login	Successful authentication, access granted to dashboard	Pass

Test ID	Test Description	Steps	Expected Result	Status
TC003	Transaction - Deposit	1. Login successfully → 2. Select Deposit → 3. Enter amount → 4. Confirm transaction	Balance updated correctly, transaction log recorded	Pass
TC004	Transaction - Withdraw	1. Login successfully → 2. Select Withdraw → 3. Enter amount → 4. Confirm transaction	Balance deducted correctly, transaction log recorded	Pass
TC005	Balance Inquiry	1. Login successfully → 2. Select Check Balance	Correct account balance displayed	Pass

Table 7.3: Test Cases

CHAPTER-8

CONCLUSION

By fusing facial recognition and PIN verification, the Face ATM System offers a complete answer to the rising need for safe and dependable banking authentication. Through extensive testing of individual deep learning models, including CNN, DenseNet201, MobileNet, and AlexNet, and their integration into an ensemble framework, the system minimizes the risk of unauthorized access while achieving high accuracy in user verification. Its use of dual Gabor filters for sophisticated image preprocessing, which greatly improves facial feature extraction, is a major innovation. When designing the Face ATM system, security, usability, and practical dependability were given top priority. The system provides a simple and user-friendly web interface through which customers can manage their profiles, log in securely, and access different banking services with ease. Once a user is authenticated, session management runs in the background to ensure their access remains safe and uninterrupted throughout the interaction.

What makes this project stand out is how it balances academic research with real banking needs. The interface is secure enough for financial institutions, yet simple enough for everyday users. It has also been designed with future growth in mind—features like video-based face authentication can be added later without the need to completely rebuild the system.

CHAPTER-9

SCOPE FOR FURTHER ENHANCEMENT

While the current **Face ATM System** demonstrates reliable authentication and secure transaction capabilities, several enhancements can further improve performance, usability, and security:

To further elevate the capabilities of the Face ATM System, several technical enhancements have been proposed that focus on both model sophistication and processing efficiency. To enhance the system's facial recognition capabilities, more recent modelling technologies such as Vision Transformers (ViT) and Swin Transformers can be incorporated. These models are renowned for their capacity to highlight significant aspects of a picture, which increases accuracy. Because efficient .Net models intelligently balance speed and performance, they are also well suited for real-time use in banking settings.

Security and performance can also be enhanced by adding real-time video verification, where the system scans face frame by frame instead of relying on just a single snapshot. In busy banking locations, the system could use batch processing to handle multiple users at once. With GPU acceleration through CUDA, it would still deliver fast responses even under heavy loads. On the infrastructure side, moving away from simple storage formats like Excel or CSV to powerful databases such as PostgreSQL or MongoDB would greatly improve data integrity and search speed. Hosting the system on cloud platforms like AWS, Azure, or Google Cloud would make it scalable and always available. To further optimize performance, caching solutions like Redis or Memcached could minimize delays, while load balancing would distribute traffic smoothly across ATMs and online banking portals.

BIBLIOGRAPHY

- [1] Kim, E. and Cho, S., 2021. Exposing fake faces through deep neural networks combining content and trace feature extractors. *IEEE Access*, 9, pp.123493–123503.
- [2] Jain, A.K., Ross, A. and Pankanti, S., 2006. Biometrics: a tool for information security. *IEEE Transactions on Information Forensics and Security*, 1(2), pp.125–143.
- [3] Jain, A.K., Ross, A. and Prabhakar, S., 2004. An introduction to biometric recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1), pp.4–20.
- [4] Li, H., Jain, A.K., and Ross, A., 2019. Face recognition for banking applications: Challenges and solutions. *IEEE Transactions on Information Forensics and Security*, 14(10), pp.2630–2644.
- [5] Rattani, A., Gianassi, L., and Choraś, M., 2020. Deep learning-based face recognition in ATMs: Methods and security considerations. *Journal of Ambient Intelligence and Humanized Computing*, 11(8), pp.3305–3321.
- [6] Zhang, K., Zhang, Z., Li, Z., and Qiao, Y., 2016. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10), pp.1499–1503.
- [7] Li, Y., Chang, M.C., and Lyu, S., 2018. I know what you saw last summer: Privacy-preserving face recognition for secure banking transactions. *Proceedings of the ACM on Computer and Communications Security*, 1(1), pp.1–13.
- [8] Jain, A., Arora, D., Bali, R., and Sinha, D., 2021. Secure authentication for banking using face recognition. *Journal of Informatics Electrical and Electronics Engineering*, 2(2), pp.1–8.
- [9] Achyuthananda, A., 2021. Bank transaction using facial identification system. *Sathyabama Institute of Science and Technology*, Bachelor of Engineering Thesis, pp.1–30.
- [10] Parkhi, O.M., Vedaldi, A., and Zisserman, A., 2015. Deep face recognition. *British Machine Vision Conference (BMVC)*, pp.1–12.
- [11] Schroff, F., Kalenichenko, D., and Philbin, J., 2015. FaceNet: A unified embedding for face recognition and clustering. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp.815–823.
- [12] Taigman, Y., Yang, M., Ranzato, M.A., and Wolf, L., 2014. DeepFace: Closing the gap to human-level performance in face verification.